Q .1 Problem Statement:

Write a generic function template named findMinimum in C++ that takes an array of any data type T and its size n as arguments. The function should return the minimum element present in the array.

#include <iostream>

using namespace std;

template <typename T>

T findMinimum(T arr[], int n) {

if (n <= 0) {

throw std::invalid\_argument("Array size must be greater than 0");

}

T minElement = arr[0];

for (int i = 1; i < n; ++i) {

if (arr[i] < minElement) {

minElement = arr[i];

}

}

return minElement;

}

int main() {

int intArr[] = {41, 12, 28, 12, 11};

double doubleArr[] = {4.5, 3.1, 8.5, 6.3,2.7};

char charArr[] = {'d', 'i', 'k', 's', 'h','a'};

int intSize = sizeof(intArr) / sizeof(intArr[0]);

int doubleSize = sizeof(doubleArr) / sizeof(doubleArr[0]);

int charSize = sizeof(charArr) / sizeof(charArr[0]);

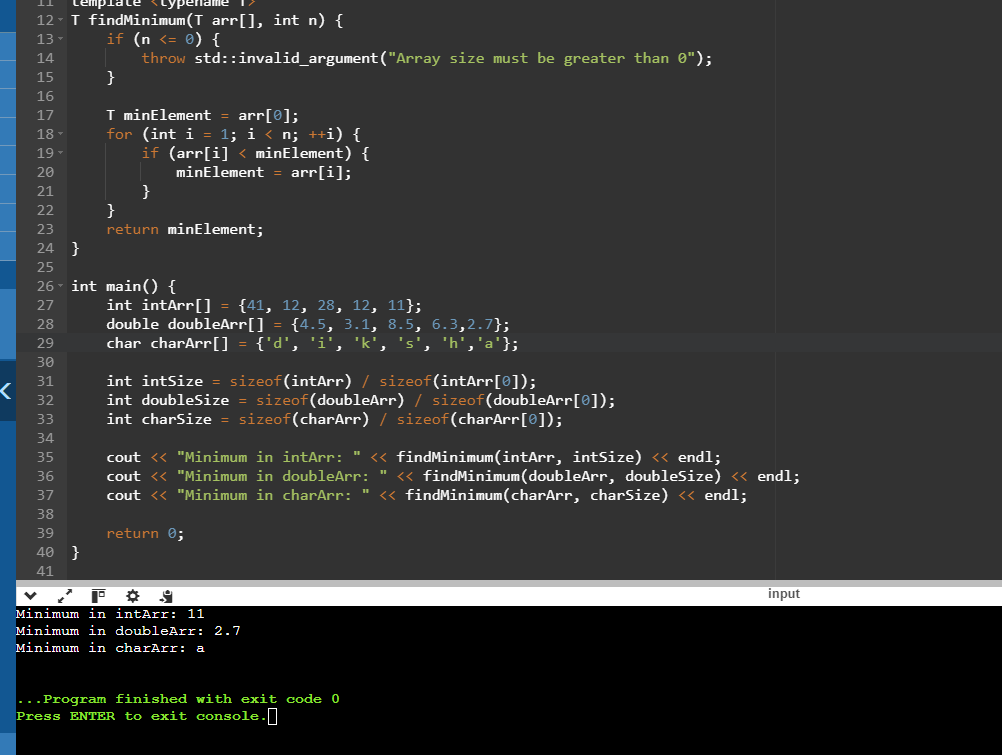
cout << "Minimum in intArr: " << findMinimum(intArr, intSize) << endl;

cout << "Minimum in doubleArr: " << findMinimum(doubleArr, doubleSize) << endl;

cout << "Minimum in charArr: " << findMinimum(charArr, charSize) << endl;

return 0;

}



1. Swap Elements:

Problem: Write a function template swap that takes two pointers to variables of any data type T and swaps their values.

Constraints: The function should only modify the values pointed to by the arguments, not the arguments themselves (pass by reference).

#include <iostream>

#include <stdexcept>

using namespace std;

template <typename T>

T findMinimum(const T\* array, size\_t n) {

if (n == 0) {

throw std::invalid\_argument("Array size must be greater than 0");

}

T minElement = array[0];

for (size\_t i = 1; i < n; ++i) {

if (array[i] < minElement) {

minElement = array[i];

}

}

return minElement;

}

template <typename T>

void swap(T\* a, T\* b) {

T temp = \*a;

\*b = temp;

}

int main(){

int x = 90;

int y = 45;

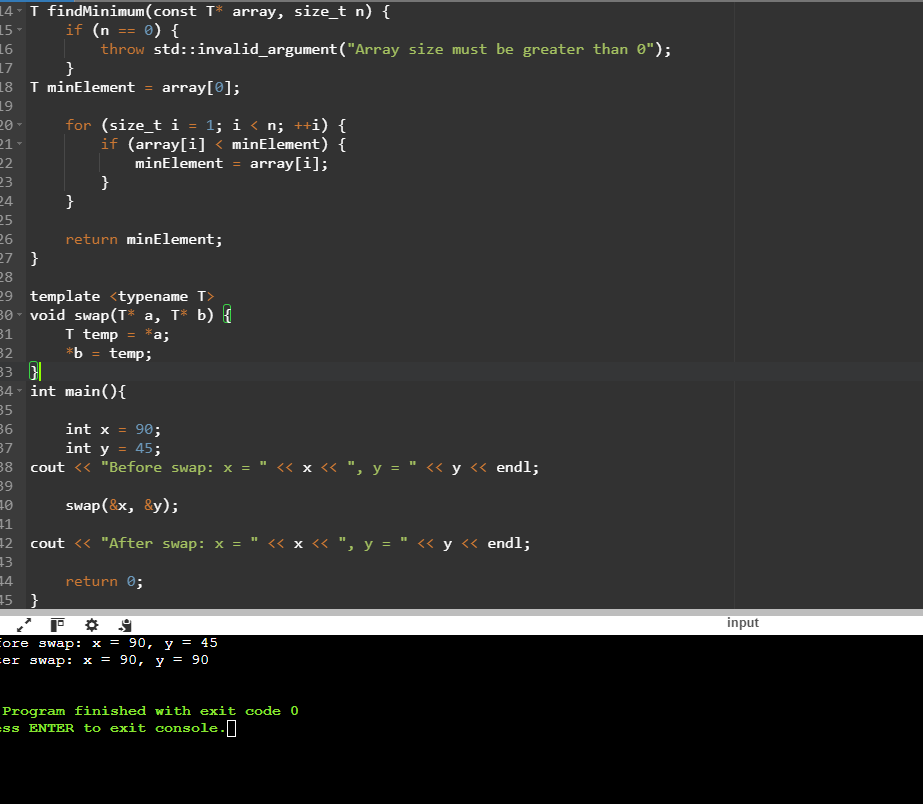
cout << "Before swap: x = " << x << ", y = " << y << endl;

swap(&x, &y);

cout << "After swap: x = " << x << ", y = " << y << endl;

return 0;

}



2. Find Maximum:

Problem: Similar to findMinimum, create a function template findMaximum that returns the maximum element in an array of any data type T.

#include <iostream>

#include <stdexcept>

template <typename T>

T findMaximum(T arr[], int n) {

if (n <= 0) {

throw std::invalid\_argument("Array size must be greater than zero.");

}

T max = arr[0];

for (int i = 1; i < n; ++i)w {

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

int main() {

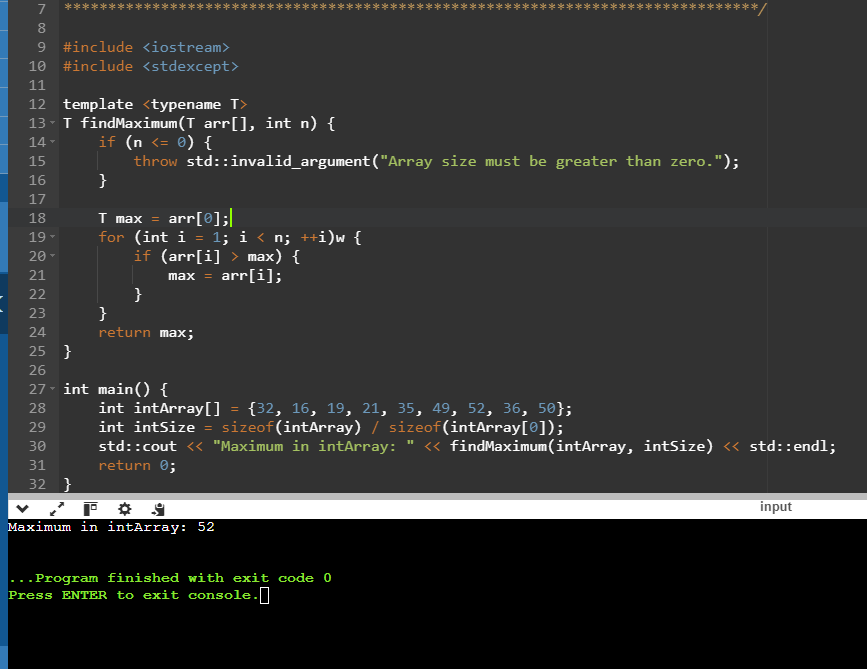
int intArray[] = {32, 16, 19, 21, 35, 49, 52, 36, 50};

int intSize = sizeof(intArray) / sizeof(intArray[0]);

std::cout << "Maximum in intArray: " << findMaximum(intArray, intSize) << std::endl;

return 0;

}



Design a generic data processing library using class and function templates in C++. This library should be able to handle various data types (e.g., integers, floats, strings) without code duplication.

Requirements:

Create a class template named DataContainer that can hold elements of any data type specified during instantiation.

Implement member functions for DataContainer:

DataContainer(size\_t size): Constructor to initialize the container with a specific size.

T& operator[](size\_t index): Overloaded subscript operator to access elements.

void printAll(): Prints all elements of the container.

Create a function template named swap that takes two DataContainer objects as arguments and swaps their elements.

Ensure proper memory management using appropriate constructors and destructors.

Coding Practice Questions:

Implement the DataContainer class template:

Define the template parameter to specify the data type.

Use an array or a vector internally to store the elements.

Implement the constructor, subscript operator, and printAll function as described in the requirements.

Implement the swap function template:

Take two DataContainer objects as arguments.

Use a loop or recursion to iterate over corresponding elements and swap their values.

Consider potential edge cases (e.g., containers of different sizes).

Write a main function to demonstrate the library:

Create instances of DataContainer for different data types (e.g., int, float, string).

Populate the containers with sample data.

Call printAll on each container to verify its contents.

Use the swap function to swap elements between containers of the same type.

Print the containers again to confirm the swap.

Enhance the DataContainer class:

Add member functions for:

size(): Returns the current size of the container.

push\_back(const T& value): Appends an element to the back of the container (dynamically resize if necessary).

Modify the constructor to accept an optional initial size (default to 0).

Explore advanced functionalities (optional):

Implement a class template for linked lists or binary search trees, leveraging the DataContainer class.

Create function templates for generic sorting algorithms (e.g., bubble sort, selection sort).

#include <iostream>

#include <vector>

#include <string>

template <typename T>

class DataContainer {

public:

DataContainer(size\_t size = 0) : data(size) {}

T& operator[](size\_t index) {

return data[index];

}

void printAll() const {

for (const auto& element : data) {

std::cout << element << " ";

}

std::cout << std::endl;

}

size\_t size() const {

return data.size();

}

void push\_back(const T& value) {

data.push\_back(value);

}

private:

std::vector<T> data;

};

template <typename T>

void swap(DataContainer<T>& a, DataContainer<T>& b) {

if (a.size() != b.size()) {

throw std::invalid\_argument("Containers must have the same size to swap");

}

for (size\_t i = 0; i < a.size(); ++i) {

std::swap(a[i], b[i]);

}

}

int main() {

DataContainer<int> intContainer(5);

DataContainer<float> floatContainer(5);

DataContainer<std::string> stringContainer(3);

for (size\_t i = 0; i < intContainer.size(); ++i) {

intContainer[i] = static\_cast<int>(i \* 2);

}

for (size\_t i = 0; i < floatContainer.size(); ++i) {

floatContainer[i] = static\_cast<float>(i \* 1.5);

}

stringContainer[0] = "Hello";

stringContainer[1] = "Generic";

stringContainer[2] = "World";

std::cout << "Initial contents:" << std::endl;

intContainer.printAll();

floatContainer.printAll();

stringContainer.printAll();

DataContainer<int> anotherIntContainer(5);

for (size\_t i = 0; i < anotherIntContainer.size(); ++i) {

anotherIntContainer[i] = static\_cast<int>(i \* 3);

}

std::cout << "\nBefore swap:" << std::endl;

intContainer.printAll();

anotherIntContainer.printAll();

swap(intContainer, anotherIntContainer);

std::cout << "After swap:" << std::endl;

intContainer.printAll();

anotherIntContainer.printAll();

std::cout << "\nDemonstrate push\_back:" << std::endl;

DataContainer<int> dynamicContainer;

dynamicContainer.push\_back(10);

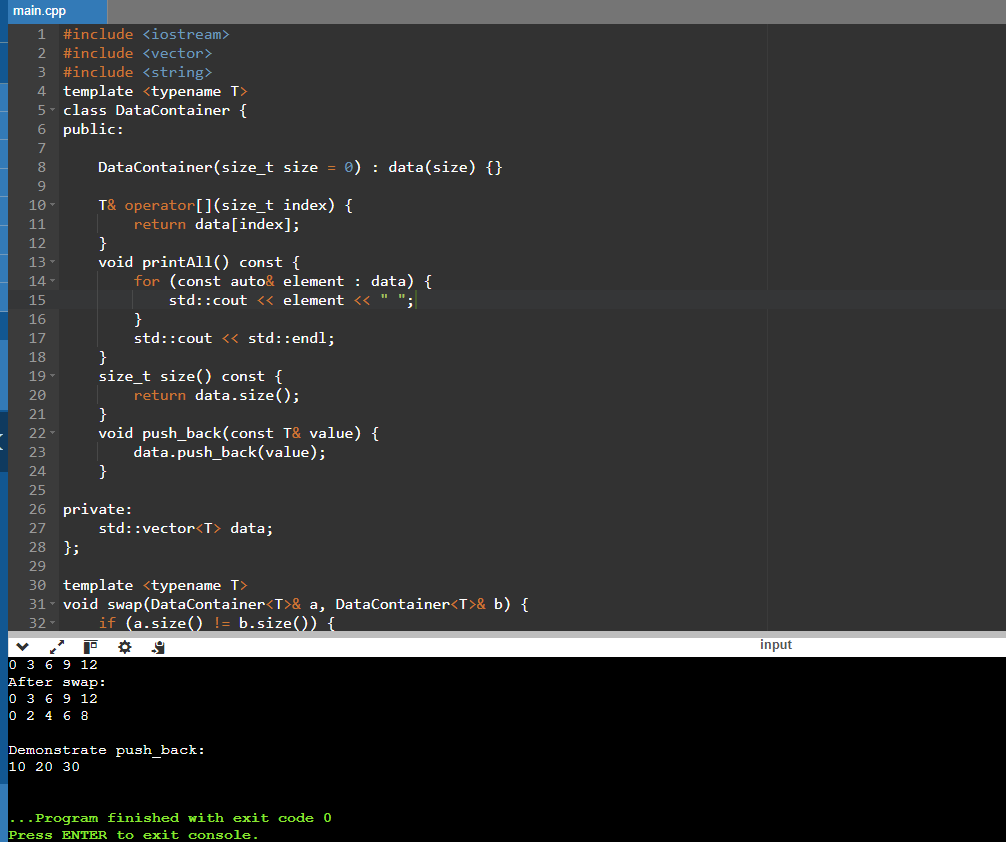
dynamicContainer.push\_back(20);

dynamicContainer.push\_back(30);

dynamicContainer.printAll();

return 0;

}



2.

#include <iostream>

#include <vector>

#include <stdexcept>

template <typename T>

class DataContainer {

public:

DataContainer(size\_t size = 0) : data(size) {}

T& operator[](size\_t index) {

return data[index];

}

void printAll() const {

for (const auto& element : data) {

std::cout << element << " ";

}

std::cout << std::endl;

}

size\_t size() const {

return data.size();

}

void push\_back(const T& value) {

data.push\_back(value);

}

private:

std::vector<T> data;

};

template <typename T>

struct Node {

T data;

Node\* next;

Node(const T& value) : data(value), next(nullptr) {}

};

template <typename T>

class LinkedList {

public:

LinkedList() : head(nullptr) {}

~LinkedList() {

clear();

}

void push\_back(const T& value) {

Node<T>\* newNode = new Node<T>(value);

if (!head) {

head = newNode;

} else {

Node<T>\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

}

}

void printAll() const {

Node<T>\* temp = head;

while (temp) {

std::cout << temp->data << " ";

temp = temp->next;

}

std::cout << std::endl;

}

void clear() {

Node<T>\* temp;

while (head) {

temp = head;

head = head->next;

delete temp;

}

}

private:

Node<T>\* head;

};

template <typename T>

void swap(DataContainer<T>& a, DataContainer<T>& b) {

if (a.size() != b.size()) {

throw std::invalid\_argument("Containers must have the same size to swap");

}

for (size\_t i = 0; i < a.size(); ++i) {

std::swap(a[i], b[i]);

}

}

// Bubble Sort

template <typename T>

void bubbleSort(DataContainer<T>& container) {

for (size\_t i = 0; i < container.size(); ++i) {

for (size\_t j = 0; j < container.size() - i - 1; ++j) {

if (container[j] > container[j + 1]) {

std::swap(container[j], container[j + 1]);

}

}

}

}

template <typename T>

void selectionSort(DataContainer<T>& container) {

for (size\_t i = 0; i < container.size(); ++i) {

size\_t minIndex = i;

for (size\_t j = i + 1; j < container.size(); ++j) {

if (container[j] < container[minIndex]) {

minIndex = j;

}

}

std::swap(container[i], container[minIndex]);

}

}

int main() {

LinkedList<int> intList;

intList.push\_back(5);

intList.push\_back(3);

intList.push\_back(8);

intList.printAll();

DataContainer<int> intContainer(5);

for (size\_t i = 0; i < intContainer.size(); ++i) {

intContainer[i] = static\_cast<int>(i \* 2);

}

std::cout << "Before sorting: ";

intContainer.printAll();

// Bubble sort

bubbleSort(intContainer);

std::cout << "After bubble sort: ";

intContainer.printAll();

// Selection sort

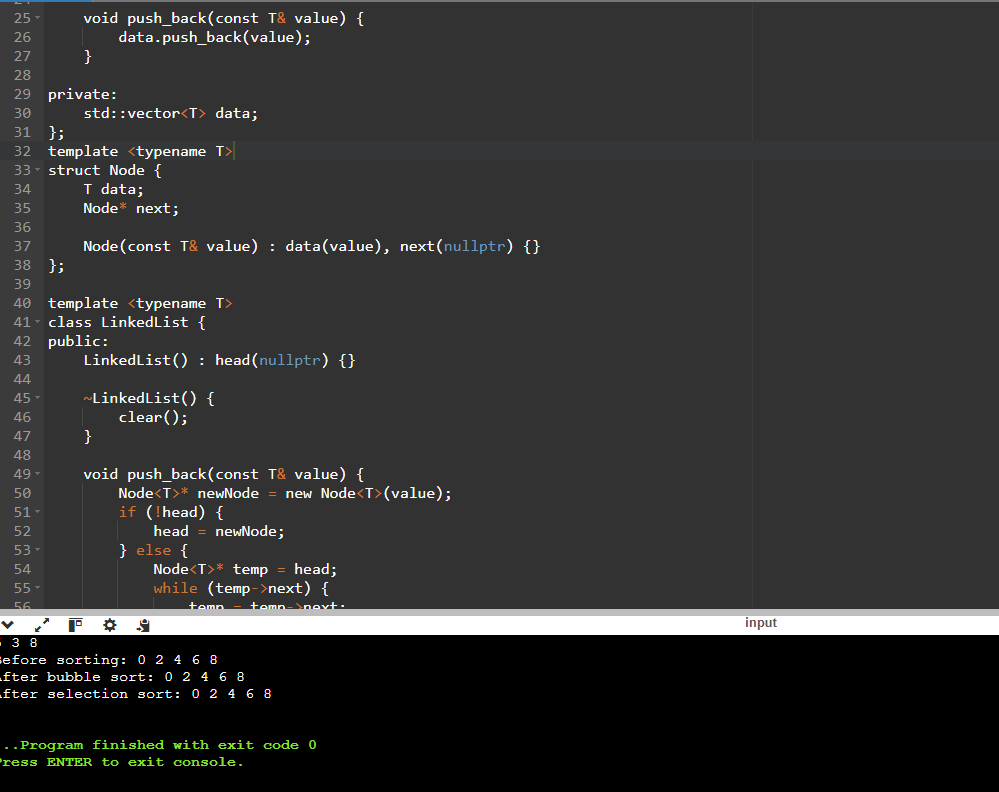
selectionSort(intContainer);

std::cout << "After selection sort: ";

intContainer.printAll();

return 0;

}



#include<iostream>

using namespace std;

//A generic smart pointer class

template <class T>

class Smartpointer {

T \*p;

public:

Smartpointer(T \*ptr = NULL){

p = ptr;

}

//Destructor

~Smartpointer()

{

delete(p);

}

//overloading dereferencing operator

T & operator \* () {

return \*p;

}

//Overloading arrow operator so that members of T can be accessed

//like a pointer

T \* operator -> () {

return p;

}

};

int main() {

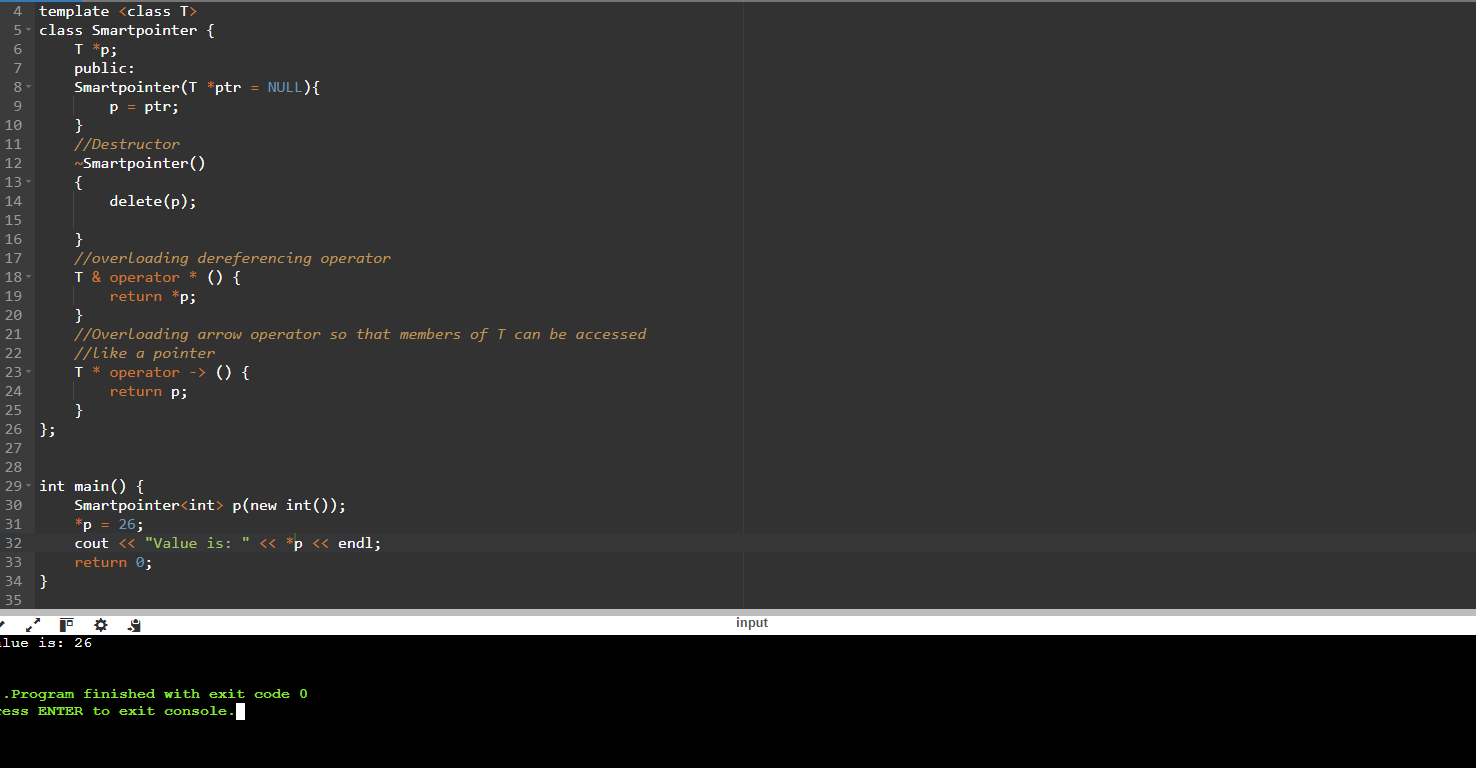
Smartpointer<int> p(new int());

\*p = 26;

cout << "Value is: " << \*p << endl;

return 0;

}



Inventory Management System:

Problem: Design a system to manage inventory for various products. Each product might have different attributes (name, price, quantity) and potentially unique functionalities (e.g., perishable items with an expiry date).

#include <iostream>

#include <vector>

#include <string>

class Product {

protected:

int id;

std::string name;

double price;

int quantity;

public:

Product(int id, const std::string& name, double price, int quantity)

: id(id), name(name), price(price), quantity(quantity) {}

virtual ~Product() {}

virtual void display() const = 0;

virtual void updateQuantity(int quantity) = 0;

virtual void updatePrice(double price) = 0;

};

// Derived class for perishable products

class PerishableProduct : public Product {

private:

std::string expiryDate;

public:

PerishableProduct(int id, const std::string& name, double price, int quantity, const std::string& expiryDate)

: Product(id, name, price, quantity), expiryDate(expiryDate) {}

void display() const override {

std::cout << "ID: " << id << ", Name: " << name << ", Price: " << price

<< ", Quantity: " << quantity << ", Expiry Date: " << expiryDate << std::endl;

}

void updateQuantity(int quantity) override {

this->quantity = quantity;

}

void updatePrice(double price) override {

this->price = price;

}

};

// Inventory class managing a collection of products

class Inventory {

private:

std::vector<Product\*> products;

public:

~Inventory() {

for (auto product : products) {

delete product;

}

}

void addProduct(Product\* product) {

products.push\_back(product);

}

void listProducts() const {

for (const auto& product : products) {

product->display();

}

}

};

int main() {

Inventory inventory;

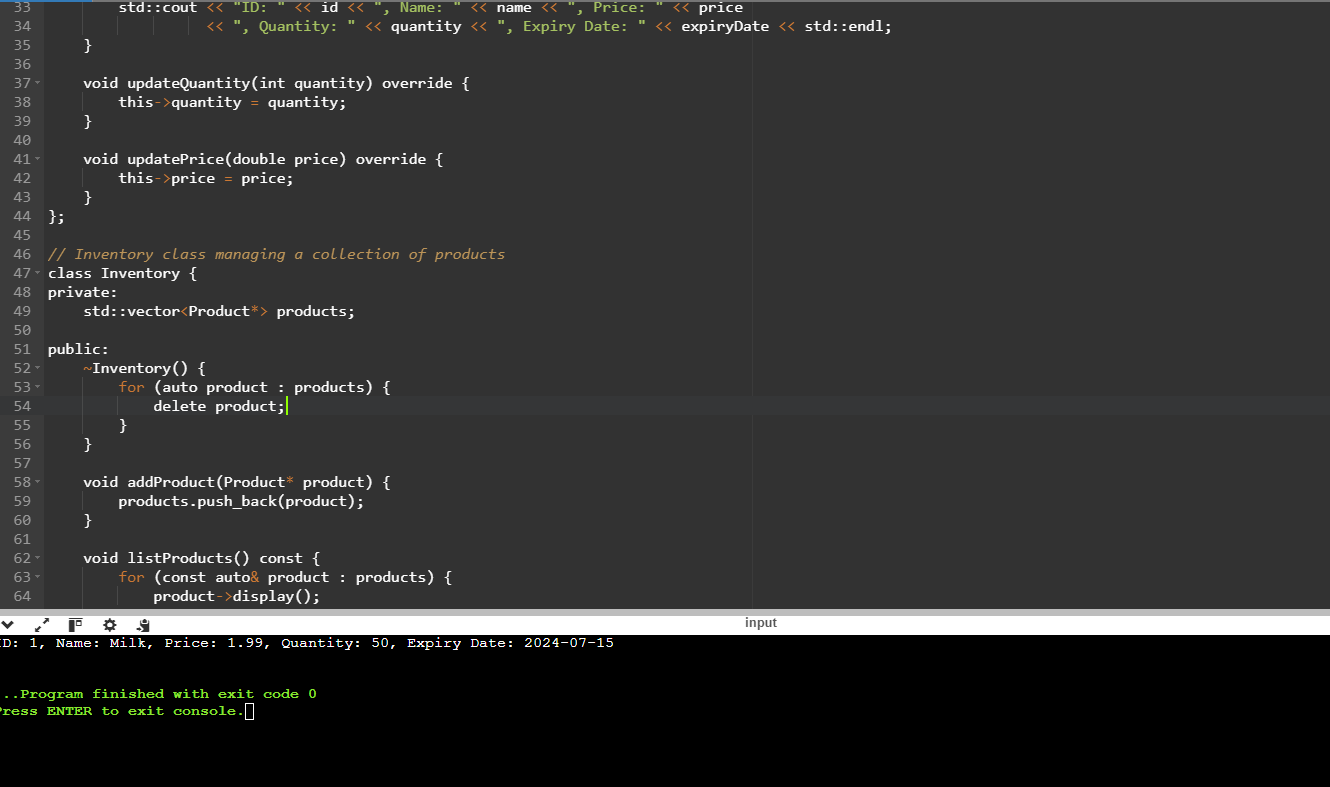
Product\* p1 = new PerishableProduct(1, "Milk", 1.99, 50, "2024-07-15");

inventory.addProduct(p1);

inventory.listProducts();

return 0;

}



In object-oriented programming with C++, abstract classes are a valuable tool for defining common interfaces and behaviors for a group of related classes. However, directly creating objects from an abstract class is not possible. This problem statement explores how abstract classes are used to enforce a design pattern and promote code reusability.

use abstract classes and polymorphism in C++ for calculating the areas of various shapes.

#include <iostream>

#include <cmath>

// Abstract base class

class Shape {

public:

virtual ~Shape() {}

// Pure virtual function for calculating the area

virtual double calculateArea() const = 0;

// Virtual function for displaying shape details

virtual void display() const = 0;

};

// Derived class for Circle

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

double calculateArea() const override {

return M\_PI \* radius \* radius;

}

void display() const override {

std::cout << "Circle with radius: " << radius << std::endl;

}

};

// Derived class for Rectangle

class Rectangle : public Shape {

private:

double width;

double height;

public:

Rectangle(double w, double h) : width(w), height(h) {}

double calculateArea() const override {

return width \* height;

}

void display() const override {

std::cout << "Rectangle with width: " << width << " and height: " << height << std::endl;

}

};

// Derived class for Triangle

class Triangle : public Shape {

private:

double base;

double height;

public:

Triangle(double b, double h) : base(b), height(h) {}

double calculateArea() const override {

return 0.5 \* base \* height;

}

void display() const override {

std::cout << "Triangle with base: " << base << " and height: " << height << std::endl;

}

};

// Function to display the area of any shape

void displayArea(const Shape& shape) {

shape.display();

std::cout << "Area: " << shape.calculateArea() << std::endl;

}

int main() {

Circle circle(7.0);

Rectangle rectangle(2.0, 1.0);

Triangle triangle(2.0, 7.0);

// Polymorphic behavior

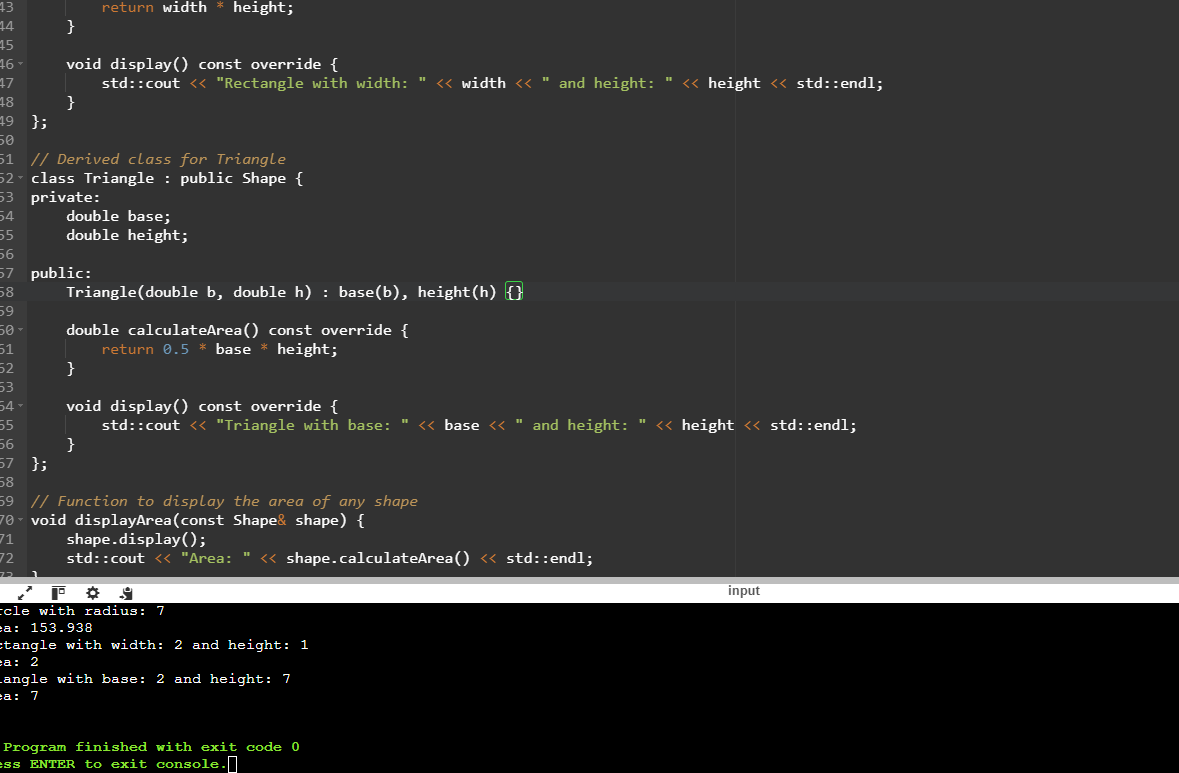
displayArea(circle);

displayArea(rectangle);

displayArea(triangle);

return 0;

}



Create your Own verion of smart pointer.

#include <iostream>

template <typename T>

class SmartPointer {

private:

T \*ptr;

size\_t \*refCount;

public:

SmartPointer(T \*p = nullptr) : ptr(p), refCount(new size\_t(1)) {}

~SmartPointer() {

decrementRefCount();

}

SmartPointer(const SmartPointer<T> &other) : ptr(other.ptr), refCount(other.refCount) {

(\*refCount)++;

}

SmartPointer<T>& operator=(const SmartPointer<T> &other) {

if (this != &other) {

decrementRefCount();

ptr = other.ptr;

refCount = other.refCount;

(\*refCount)++;

}

return \*this;

}

T& operator\*() const {

return \*ptr;

}

T\* operator->() const {

return ptr;

}

size\_t use\_count() const {

return \*refCount;

}

private:

void decrementRefCount() {

if (--(\*refCount) == 0) {

delete ptr;

delete refCount;

}

}

};

int main() {

SmartPointer<int> sp1(new int(20));

std::cout << "sp1: " << \*sp1 << ", ref count: " << sp1.use\_count() << std::endl;

{

SmartPointer<int> sp2 = sp1;

std::cout << "sp2: " << \*sp2 << ", ref count: " << sp2.use\_count() << std::endl;

}

std::cout << "sp1 after sp2 scope: " << \*sp1 << ", ref count: " << sp1.use\_count() << std::endl;

SmartPointer<int> sp3(new int(40));

sp1 = sp3;

std::cout << "sp1 after assignment: " << \*sp1 << ", ref count: " << sp1.use\_count() << std::endl;

return 0;

}

